FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Making C++ Ready for Algorithmic Skeletons

Jörg Striegnitz

FZJ-ZAM-IB-2000-08

September 2000

(letzte Änderung: 14.09.2000)

Making C++ Ready for Algorithmic Skeletons

Jörg Striegnitz

Central Institute for Applied Mathematics
Research Centre Juelich
Federal Republic of Germany
J.Striegnitz@fz-juelich.de,
WWW project page: http://www.fz-juelich.de/zam/FACT

Abstract. Many authors have proposed the use of algorithmic skeletons as a high level, machine independent means of developing parallel applications. Since now their implementation and use was restricted to either functional-, or some sophisticated imperative languages. In this paper we will discuss how far C++ supports the integration of algorithmic skeletons and identify *currying* as the only missing feature. We will show how this gap can be closed, by integrating currying into C++ through code that is compliant with the ANSI/ISO standard, thus, by using the language itself instead of extending it. We will prove that our method does not yield any runtime penalties if a highly optimizing C++ compiler is used and, therefore, is competitive with existing sophisticated languages.

1 Introduction

Algorithmic skeletons represent an approach to parallel programming. The basic idea is to replace explicit parallel programming (e.g. using a parallel language, or a message passing library), by the selection and instantiation of a variety of pre-packaged parallel algorithmic forms known as *skeletons* [1].

Usually skeletons are embedded into a sequential programming language, thus, being the only source of parallelism for a program. Most of them, like for instance *map*, *farm*, and *divide and conquer*, are implemented as polymorphic higher order functions [3]. Both features are common in functional programming languages and, therefore, most languages with skeletons build upon a functional host [1, 2]. There are also some sophisticated imperative languages like SKIL [4] which is an extension of C.

2 Functional Features in C++

In [3] at least three features are identified that a programming language has to provide in order to be ready for algorithmic skeletons:

- *polymorphic types*,
- *higher order functions*,
- *partial application*.

In the following sections we will discuss these features in more depth and show how far C++ already supports them and discover what is still missing.

2.1 Polymorphic Types

C++ supports polymorphic types through templates. A template definition contains a list of type variables, followed by the definition of a function, a class member function, or a class, which may make use of those type variables. For instance, a function template to calculate the average of two values may look like this:

```
template <typename T>
T average(T a,T b) {
   return (a + b) / 2;
}
```

Instantiation of templates usually is done automatically by the C++ compiler. So, passing two integers to `average` will enforce instantiation of `int average(int,int)`. Notice that `average` requires summation- and division operators to be defined for the type represented by `T`. Fortunately, C++ allows the definition of almost all important operators for user-defined classes (e.g. `+, -, *, /, <, >, <=, >=`).

2.2 Higher Order Functions

Higher order functions are functions with functional arguments and/or functional results. Indeed the implementation of higher order functions is not only possible with C++, but also the Standard Template Library (STL) [5, 6] – as part of the C++ standard – already contains a lot of them (e.g. `for_each`, `transform`). Higher order functions easily can be expressed through the use of templates. Consider for example the following function template:

```
template <typename ARG,
          typename RESULT,
          typename UNARYOP,
          typename BINARYOP>
RESULT apply_op(ARG argument,UNARYOP unary,BINARYOP binary) {
  return binary(unary(argument),argument);
}
```

`UNARYOP` and `BINARYOP` respectively, represent any C++ type that supports the function call syntax, e.g. pointers to functions, or classes that provide a parentheses operator (often called *functional objects* or *functors*):

```
class MyFunctionalClass {
  ...
  inline int operator()(int a, int b) {
    return a + b;
  }
  ...
};
```

The keyword `inline`[1], used in the above code, instructs the compiler that code produced for the following function definition should be inlined – if possible. For that reason functional objects could be expected to have no impact on the performance of a program.

2.3 Partial Application

With partial application the first $k < n$ arguments of an n-ary function get bound to specific values, thereby yielding a function of order $n - k$.

Binding either the first or the second argument of a binary function is possible with C++ (`bind1st`,`bind2nd`). Although partial application of binary functions is supported this way, its application looks complicated. This is due to `bind1st` expecting its first argument to be a functor. Thus, before partial application of an ordinary C++ function may happen, it has to be converted into a functor by a calling `ptr_fun`. Consider for example a given C++ function `int sum(int a,int b)`. Partially applying `sum` to `4` looks like this in standard C++:

```
bind1st( ptr_fun(sum) , 4 )
```

This call returns a functional object that offers a unary parenthesis operator that gets a single integer and returns its value incremented by four. For that reason you may call `bind1st( ptr_fun(sum) , 4 )(38)` to retrieve 42.

In functional programming languages partial application is naturally possible, because functions are represented in their curried form. Currying has its roots in the mathematical study of functions where it has been shown that it is sufficient to restrict attention to unary functions: every function $f : A_1 \times \cdots \times A_n \to R$ can be turned into a unary function $g : A_1 \to \cdots \to A_n \to R$, with $\to$ being associative to the right. g then is called the curried form of f.

Passing a single argument to g leads to another unary function to which we may pass a value of type A_2 to obtain another unary function to which we may pass a value of type A_3 to retrieve yet another unary function to which ... and so on, until we get a unary function which acts on a value type A_n and returns a value of type R. Of course $g(a_1) \cdots (a_n)$ should produce the same result as $f(a_1, \cdots, a_n)$.

Now, although g is unary, we may think of it as being n-ary, but with the special capability to get its arguments one at a time, because calling $g(a_1)...(a_k)$, $k < n$ yields a valid result – a unary function. But how shall this unary function look like ? It is the curried form of a function h of order $n - k$ which is defined as follows:

$$h : A_{k+1} \times \cdots \times A_n \to R$$
$$h(a_{k+1}, \cdots a_n) = f(\underbrace{a_1, \cdots, a_k}_{constant}, a_{k+1} \cdots, a_n)$$

[1] Not necessary at this point because all members defined in class scope are *inline* per default

Thus, we retrieve the curried form of f, whereas the first k arguments have been bound to $a_1, \cdots, a_k$.

All in all C++ has all features necessary to support algorithmic skeletons – except currying of arbitrary functions. In the next section we will discuss how to integrate currying into C++ without being in need of changing the language itself.

3 Our New Concept: Functional Functors

Our primary goal was to develop a function called `curry` that takes a C++ function and returns its curried representation. Before discussing this function in detail we need to consider how curried functions should be represented.

We distinguish three possibilities to represent a function in C++:

1. C++ function,
2. functional objects – functors,
3. and our new concept: *functional functors* (in short: `f-functors`)

A C++ function is a function with C++ linkage or a static class member function. Functional objects are classes which provide a parentheses operator that is compatible with the signature of the function they should represent. f-functors are special functional objects. Like functional objects they also offer a parentheses operator, but in contrast to them, this operator always has the signature of a unary function. Moreover, an f-functor usually is more general than a functor, because it has to be supplied with a computational rule.

Functional functors are implemented as class templates, generalizing argument type(s), return type, and the type of the computational rule. Here is the definition of a functional functor, used to represent curried unary functions:

```
template <typename ARG,typename RESULT,     // Signature
          typename UNARYOP=RESULT (*)(ARG)> // Comp. rule
class FUNC1 {                               // (CR)
private:
  const UNARYOP op;
public:                                  // Need CR to initialize
  FUNC1( const UNARYOP &op_ ) : op (op_) {}
  inline RESULT operator()( ARG a ) const {
    return op( a );                      // Hand off argument to
  }                                      // computational rule
};
```

Once again notice that `UNARYOP` may either be a pointer to a C++ function, a functional object, or a functional functor. In the previous given definition it defaults to be a pointer to function because we expect this to be the most common situation. Since a unary function and its curried form are identical, nothing special has to be done: `FUNC1`'s parentheses operator just hands off its argument to the computational rule represented by `op`.

A functional functor to represent binary functions is a little bit more complicated. Consider a binary function f and its curried form g:

$$f : A_1 \times A_2 \rightarrow R \; ; \; curry(f) = g : A_1 \rightarrow (A_2 \rightarrow R)$$

Applying g to a single argument $a_1 \in A_1$ yields a unary function of type $(A_2 \rightarrow R)$. In our model unary functions are represented through the class template `FUNC1`, thus, an f-functor that represents g should provide a unary parentheses operator that returns an object of type `FUNC1`. Since `FUNC1` is a template, we have to think about the types to use during instantiation. So far, argument- and result type are known, because they are dictated by g's signature. But what about the type of the computational rule ?

The computational rule has to be: take an argument of type A_2 and use it together with the parameter a_1 – that already has been passed to g – and return the result of the application of f to both arguments.

Such a computational rule only makes sense in conjunction with `FUNC2`, thus, the functional object representing it (`PAF`), is defined in the local scope of `FUNC2`:

```
template <typename ARG1, typename ARG2, typename RESULT,
          typename BINARYOP = RESULT (*)(ARG1,ARG2)>
class FUNC2 {
private:
  const BINARYOP op;
public:
  FUNC2( const BINARYOP &op_ ) : op (op_) {}

  struct PAF {           // Partial Application Functor
    const BINARYOP op;   // Memorize operation
    ARG1           a1;   // Store argument already passed to op
    PAF(const BINARYOP & op_,ARG1 a1_) : op(op_),a1(a1_) {}
    inline RESULT operator()(ARG2 a2_) const {
      return op(a1,a2_); // Make use of memorized argument
    }
  };

  FUNC1<ARG2,RESULT,PAF> operator()( ARG1 a1 ) const {
    // Notice that two temporaries are created here:
    //  - PAF
    //  - FUNC1<ARG2,RESULT,PAF>
    return FUNC1<ARG2,RESULT,PAF>(  PAF(op,a1)  );
  }
};
```

At this point we can define a functional functor to represent our `sum` function and make use of it:

```
FUNC2<int,int,int> fsum(sum);  // Use default type for CR:
result = fsum(1)(2);           // int (*)(int,int)
```

`fsum(1)(2)` now is a valid C++ expression. `fsum(1)` returns an objects of type `FUNC1<int,int, FUNC2<int,int,int>::PAF>` which indeed offers a unary parentheses operator that acts on an integer.

Developing functional functors for functions of arbitrary dimension is straight forward now. It even is a little bit boring, and thus, we developed a code-generating tool to do it for us. The code generator is supplied with the largest dimension a function to curry may have and produces an appropriate C++ header file.

Dealing with functions of larger dimension, you have to struggle with plenty of parentheses. For instance consider a function `sum3` which acts on three integer values and its curried form, represented by `fsum3`, which is of type `FUNC3<int, int, int, int>`. Passing all arguments to `fsum3`, you have to write `fsum3(1)(2)(3)`. To avoid this, we equipped a functional functor to represent a function of order n with n parentheses operators, so that you may use either of the following variants:

```
fsum3(1)(2)(3)
fsum3(1,2)(3)
fsum3(1)(2,3)
fsum3(1,2,3)
```

Although the definition of an f-functor looks quite natural, it usually is not necessary, since partial application mostly occurs when passing a function to a higher order function, thus, the curried form of a function primary is used temporarily only – the curry function is still missing. Fortunately its definition is quite easy now: `curry` is implemented as function template, general in the signature of the function to curry. It takes a pointer to a C++ function and returns an appropriate functional functor object. The `curry` function used for binary functions looks like this:

```
// Returns functional functor to represent binary function that
// gets its computational rule from a pointer to a C++ function
// which is given by the argument passed to curry -- cfunc.
template <typename ARG1,typename ARG2,typename RESULT>
inline                          // Avoid a function call to curry
FUNC2<ARG1,ARG2,RESULT, RESULT (*)(ARG1,ARG2)>
  curry(  RESULT (*cfunc)(ARG1,ARG2)  ) {
  return FUNC2<ARG1,ARG2,RESULT,RESULT (*)(ARG1,ARG2)>( cfunc );
}
```

As with functional functors, for each dimension a function to curry may have, a suitable curry function is needed. Thus, they are created by the generator tool as well.

Now, it is possible to call `curry(sum)(2)(3)` instead of defining a functional functor first. But notice that the functional functor returned by curry gets its

computational rule from a pointer to a function. With some compilers this introduces an indirect call when invoking the computational rule, thus, revealing a possible performance penalty, because inlining is forced to stop at this point.

To avoid this we introduced a new functor type that is realized as class template as well, but whose computational rule occurs as a constant template parameter:

```
template <typename ARG1,typename ARG2,typename RESULT,
          RESULT (* cfunc)(ARG1,ARG2)> // Not a type, but
class MakeF2 {                         // a constant !
public:
  typedef FUNC2< ARG1,ARG2,RESULT, MakeF2 > ffunc_t;
  inline RESULT operator()(ARG1 a1,ARG2 a2) const {
    return cfunc(a1,a2);     // Address of cfunc is known during
  }                          // compile-time -> no penalty
};                           // causing indirection
```

Using `MakeF2` together with `FUNC2` there no longer is an indirection during evaluation, but the declaration of a functional functor becomes more complicated since it has to be supplied with a computational rule of type `MakeF2`. For instance a definition of a functional functor for the `sum` function has the following form:

```
FUNC2< int,int,int, MakeF2<int,int,int,sum> >
   fsum( MakeF2<int,int,int,sum>() );
```

Fortunately it is possible to make this more convenient by introducing yet an additional class template:

```
template <typename ARG1,typename ARG2,typename RESULT,
          RESULT (* cfunc)(ARG1,ARG2) >
class FUNC2_ptr : public
                  MakeF2<ARG1,ARG2,RESULT,cfunc>::ffunc_t {
};
```

`FUNC2_ptr` is a derivative of the functional functor type offered by `MakeF2`, which itself gets its computational rule from `MakeF2` and ,thus, you may use

```
FUNC2_ptr< int,int,int, sum>  fsum;
```

to define a functional functor for `sum`[2].

4 Portability and Performance

We have tested our library (called **FACT!** - **F**unctional **A**dditions to **C**++ through **T**emplates and Classes) on several platforms. So far we were successful with the following C++ compilers:

[2] In the original implementation all f-functors have a private default constructor and the MakeF* classes are declared as their friends. Thus, we do not need a constructor argument at this point

- KCC 3.4f (Kuck and Associates)
- GCC 2.95.2 (Free Software Foundation)
- Visual C++ 6.0 + Service Pack 3 (Microsoft)
- Workshop Pro C++ 5.0 (Sun)
- Portland Group C++ (Portland Group)
- MipsPro C++ 7.3.1.1 (SGI)

Our testing environment consists of a Sun Ultra10 (UltraSparcIIi with 333 MHz, 384 MB RAM) running under Solaris 7. To estimate the performance of the `curry` function we ran several tests. All of them are based on the following C++ function:

```
inline
int sum(const int& a,const int& b,const int& c,const int& d) {
  return a + b + c + d;
}
```

First we investigated how much time it takes to evaluate this function fifty million times, by either calling it directly, using its curried form obtained by `curry`, or using the curried form obtained through the use of `FUNC4_ptr`. When using the curried forms we did not make use of our additional parentheses operators, but called the function as if it is a unary one, thus, we called e.g. `curry(sum)(a)(b)(c)(d)`.

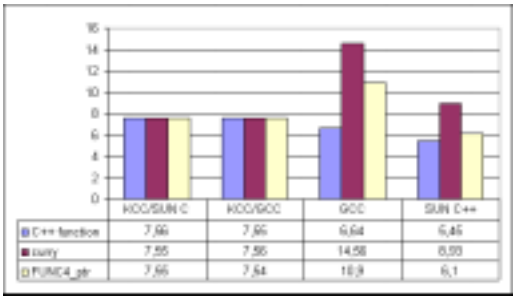

Fig. 1. Calling a C++ function and its curried variants. The chart shows the absolute time to execute each kind of evaluation fifty million times. Using a highly optimizing C++ compiler (like Kuck and Associates KCC) there is no impact on the performance if using our new concepts.

Since using curried variants of a function without partial application is quite unusual, we ran another two tests. In these tests we tried to discover the overhead when passing a partially applied function to another higher order function and distinguished whether this higher order function has been declared `inline` or not.

To get an idea on how competitive we are with existing sophisticated languages, we also run a test using SKIL [4]. SKIL is a sophisticated imperative

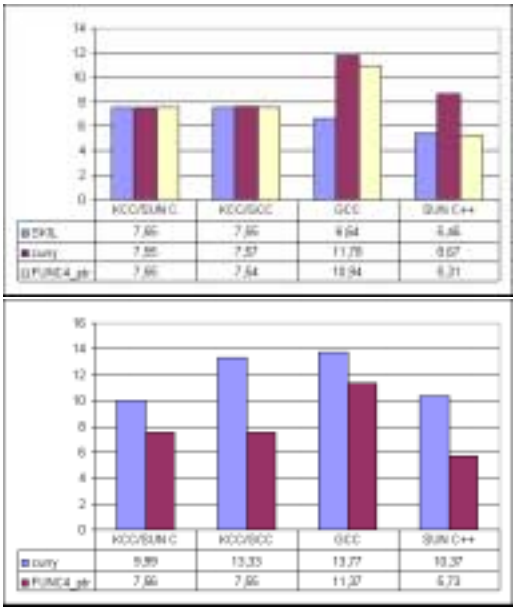

Fig. 2. Passing partially applied function to inline function (top) and to non-inline function (bottom).

language – based on C – that has support for polymorphic types, higher order functions and partial application. Since SKIL is based on C we did not distinguish whether the higher order function was declared inline or not. Figure 2 shows that using a highly optimizing C++ compiler we are in fact competitive with SKIL.

5 Conclusion and Future Work

Our aim was to figure out if it is possible to use algorithmic skeletons with C++. We identified currying as the only missing feature and demonstrated how this can be integrated into the language be relying on its own concepts. Using a highly optimizing C++ compiler it in fact is possible to use this new feature without any impact on the performance, thus, increasing the expressiveness of the language without paying any extra price. In summary C++ seems to be an ideal platform for the integration of skeletons. This even becomes more important as there are upcoming parallel versions of the Standard Template Library (STL) [7, 8]. As already mentioned the STL contains many higher order functions, especially some of them are similar to the skeletons proposed in [1] (e.g. `transform` versus `map`).

For further investigations we consider to develop our own parallel version of the STL. Some other research may be to discover whether the programming

techniques presented in [9–11] could be used to implement optimizing code transformations, as proposed in [12, 13].

Furthermore, we are currently investigating several methods to integrate lambda expressions and lazy evaluation into C++.

References

1. Darlington J., Field A.J. et al. Parallel Programming Using Skeleton Functions Proceedings of PARLE '93, LNCS 694, Springer 1993
2. Darlington J., Guo Y. et al. Functional Skeletons for Parallel Coordination Proceedings of EuroPar '95, LNCS 966, Springer 1995
3. Botorog G.H., Kuchen H. Efficient Parallel Programming with Algorithmic Skeletons Proceedings of EuroPar '96 Vol. 1, LNCS 1123, Springer 1996
4. Botorog G.H., Kuchen H. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming Proceedings of HPDC-5, IEEE Computer Society Press, 1996
5. International Standard, Programming Languages - C++, ISO/IEC: 14882, 1998
6. Austern M. Generic Programming and the STL Addison Wesley 1999
7. Silicon Graphics Computer Systems Parallelizing STL constructs http://reality.sgi.com/austern/pSTL/parallel-STL.html
8. Johnson E., Beckman P., Gannon D. HPC++: An Experiment with the Parallel Standard Template Library http://www.cs.indiana.edu/hyplan/ejohnson/papers/ics97/ics97.html
9. Veldhuizen T, Gannon D. Active Libraries: Rethinking the roles of compilers and libraries
10. Veldhuizen T. Expression Templates C++ Report Vol. 7 No. 5, June 1995
11. Veldhuizen T. Template Meta Programs C++ Report Vol. 7 No. 4 May 1995
12. Gorlatch, S. Optimizing Compositions of Scans and Reductions in Parallel Program Derivation Technical Report MIP-9711, University of Passau, Germany May 1997
13. Gorlatch, S., Pelagatti S. A Transformational Framework for Skeletal Programs: Overview and Case Study Proceedings of IPPS/SPDP '99, Springer 1999